# **08-PostgreSQL 17: Complete Tuning Guide for VACUUM & AUTOVACUUM**

[](https://medium.com/@jramcloud1?source=post_page---byline--aa36b945a7cf---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--aa36b945a7cf---------------------------------------)

Following

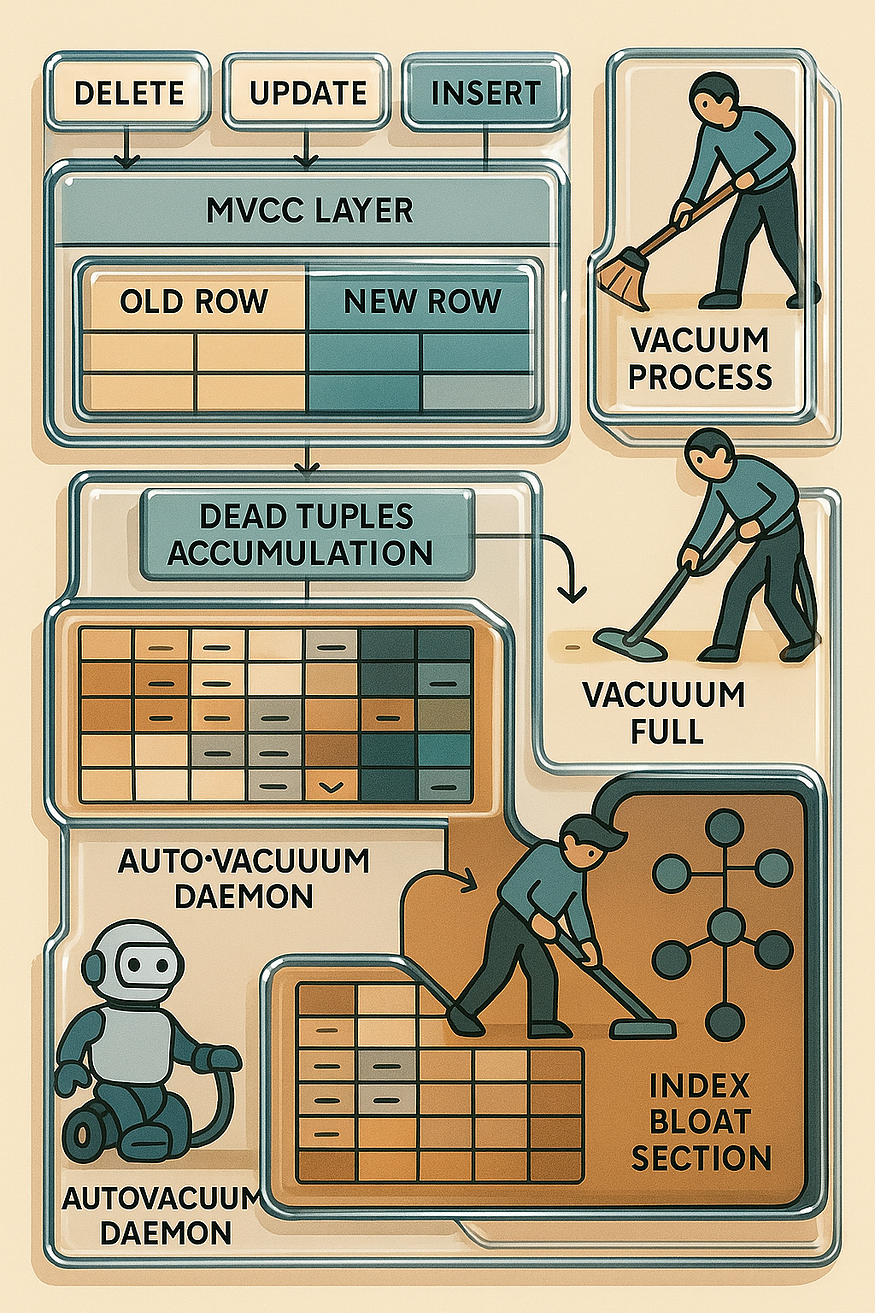
13 min read

·

Aug 31, 2025

27

Press enter or click to view image in full size



PostgreSQL’s MVCC design creates ****dead tuples**** during UPDATE/DELETE. ****VACUUM**** reclaims them; ****AUTOVACUUM**** schedules that work. Get these knobs right and you minimize bloat, avoid wraparound, and keep queries fast.

Tip: Almost every autovacuum knob has a **per-table override** via ALTER TABLE … SET ( … ). That’s how you tune hot tables aggressively without punishing the whole cluster.

## **0) Hard Prerequisites for VACUUM in PostgreSQL 17**

Before diving into VACUUM tuning itself, there are a few ****critical prerequisites**** that must be in place. Without these, even the best autovacuum settings won’t deliver good performance. Think of them as the foundation of your PostgreSQL house: if the base isn’t strong, no amount of decoration will help.

## **🔹**track\_counts = on**— Required for Autovacuum Decisions**

PostgreSQL’s autovacuum system depends on statistics about how many rows were inserted, updated, or deleted.  
These statistics are tracked only when the parameter track\_counts is set to ****on****.

* If track\_counts = off, PostgreSQL has ****no visibility**** into dead tuples.
* That means autovacuum will not know when to run, leading to table bloat, slower queries, and eventually transaction ID wraparound risks.

👉 ****Example (postgresql.conf):****

track\_counts = on

✅ ****Best Practice:**** Always keep track\_counts enabled. Disabling it is almost never safe in a production system.

📌 ****Analogy:**** It’s like keeping your car’s fuel gauge working. Without it, you’ll never know when you’re about to run out of fuel.

## **🔹**shared\_buffers**Sized Sanely — Vacuum Relies on Buffer Cache**

shared\_buffers is one of the most important PostgreSQL memory settings. It defines how much RAM PostgreSQL can use for caching table and index data.

* A ****too-small value**** means vacuum has to constantly fetch data from disk, which is slow.
* A ****too-large value**** can cause memory pressure and starve the operating system’s own disk cache.

👉 ****General guideline (PostgreSQL 17):****

* For dedicated database servers: set shared\_buffers to ****25%–40% of total RAM****.
* For mixed workloads (DB + other apps on same server): use a lower value to avoid starving other processes.

👉 ****Example (postgresql.conf):****

shared\_buffers = 8GB # on a server with 32GB RAM

✅ ****Best Practice:**** Vacuum benefits directly from a well-sized buffer cache because it avoids repeated I/O operations.

📌 ****Analogy:**** Think of shared\_buffers as the size of your desk. If your desk is too small, you keep running back to the filing cabinet (disk). If your desk is reasonably large, you can spread out and work faster.

## **🔹 Enough I/O — Vacuum Is I/O Bound**

VACUUM is not primarily a CPU-intensive process — it’s mostly ****I/O bound****. It reads tables, scans pages, and updates indexes. If your storage system is slow, vacuum performance will suffer.

* On ****fast SSDs or NVMe drives****, vacuum can keep up even with high-churn workloads.
* On ****slower spinning disks****, vacuum may lag, creating backlog and table bloat.

PostgreSQL provides ****cost-based throttling parameters**** (vacuum\_cost\_delay, vacuum\_cost\_limit) to balance vacuum work with user queries. These settings help control how aggressively vacuum consumes I/O resources.

👉 ****Example:****

vacuum\_cost\_limit = 2000  
vacuum\_cost\_delay = 5ms

This means: vacuum will pause briefly after doing a chunk of I/O work, reducing the chance it competes heavily with user queries.

✅ ****Best Practice:**** Ensure your hardware (or cloud storage) can handle sustained I/O loads, and then tune cost-based throttling according to your workload.

📌 ****Analogy:**** Imagine cleaning a very large warehouse. If your broom (I/O system) is flimsy, you’ll clean very slowly. A stronger broom (faster disks) makes the cleanup much more efficient.

## **✨ Summary**

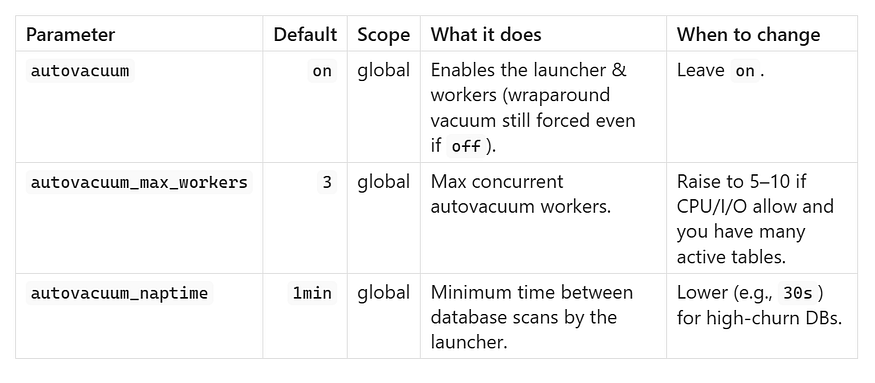
Before tuning autovacuum and vacuum in PostgreSQL 17, make sure the basics are covered:

1. ****track\_counts = on**** → Without statistics, autovacuum is blind.
2. ****Proper shared\_buffers sizing**** → A balanced buffer cache speeds up vacuum dramatically.
3. ****Sufficient I/O capacity**** → Vacuum is I/O bound, so invest in fast storage and tune cost-based throttling.

With these prerequisites in place, PostgreSQL’s vacuuming can run efficiently, keeping your database healthy and preventing bloat.

## **1) Core autovacuum controls**

Press enter or click to view image in full size



****Example****

## **🔹**autovacuum = on

This turns on PostgreSQL’s ****autovacuum launcher****, which is the background process that automatically schedules vacuuming and analyzing of tables.

* If this is set to off, PostgreSQL will not run routine autovacuum jobs.
* However, PostgreSQL will still force emergency vacuums for transaction ID wraparound protection.  
  ✅ ****Best practice:**** Always keep this enabled in production, otherwise tables will bloat and performance will degrade.

## **🔹**autovacuum\_max\_workers = 5

This defines the ****maximum number of autovacuum worker processes**** that can run at the same time.

* Default is 3, which can be too low if you have many busy tables.
* By raising it to 5, PostgreSQL can clean more tables in parallel.
* Increasing this helps large databases but also means more CPU and I/O usage.  
  ✅ ****Tip:**** Tune this based on server capacity — for example, 5–10 workers on a server with good CPU and fast storage.

## **🔹**autovacuum\_naptime = 30s

This sets how often the ****autovacuum launcher wakes up**** to check all databases for work.

* Default is 1 minute (60s).
* By lowering it to 30s, PostgreSQL checks more frequently, which helps in high-churn databases where rows are constantly updated or deleted.
* The trade-off is slightly more background activity, but tables stay cleaner.  
  ✅ ****Good for OLTP systems**** where tables accumulate dead rows quickly.

This setup makes PostgreSQL more proactive in cleaning up dead tuples, especially in workloads with frequent inserts, updates, and deletes.

autovacuum = on  
autovacuum\_max\_workers = 5  
autovacuum\_naptime = 30s

## **2) When VACUUM/ANALYZE triggers (threshold + scale factor)**

PostgreSQL decides ****when to run autovacuum**** on a table by comparing the number of ****dead tuples**** (rows that have been deleted or updated and are no longer visible) against a threshold.

That threshold is calculated with this formula:

dead\_tuples >= autovacuum\_vacuum\_threshold   
 + autovacuum\_vacuum\_scale\_factor × reltuples

Let’s explain each piece:

## **🔹**dead\_tuples

This is the number of row versions in a table that are no longer needed. They are invisible to queries but still take up space.  
👉 Autovacuum is triggered once this count passes the threshold.

## **🔹**autovacuum\_vacuum\_threshold

This is a ****fixed minimum number of dead tuples**** that must exist before autovacuum considers a table for cleanup.

* Default: 50
* Acts as a “base cost” — even very small tables won’t be vacuumed unless they have at least this many dead rows.

## **🔹**autovacuum\_vacuum\_scale\_factor

This is a ****fraction of the table size**** (reltuples = number of rows) added to the threshold.

* Default: 0.2 (20%)
* Larger tables need more dead tuples before autovacuum kicks in.

## **🔹**reltuples

This is PostgreSQL’s estimate of the total number of rows in the table, stored in system catalogs.

## **🔧 Example Calculation**

Suppose you have a table orders with ****1,000,000 rows**** (reltuples = 1,000,000).

Default settings:

autovacuum\_vacuum\_threshold = 50  
autovacuum\_vacuum\_scale\_factor = 0.2

Threshold =

50 + (0.2 × 1,000,000)   
= 50 + 200,000   
= 200,050

👉 Autovacuum will run once ****dead\_tuples ≥ 200,050****.

That means ****20% of the table**** must be dead before vacuum kicks in.

## **⚡ Why Tuning Matters**

For large OLTP tables with constant updates/deletes, waiting until 20% of rows are dead is too late — the table may bloat badly.

So you can ****lower the scale factor****:

autovacuum\_vacuum\_scale\_factor = 0.02 # 2%

Now the threshold is:

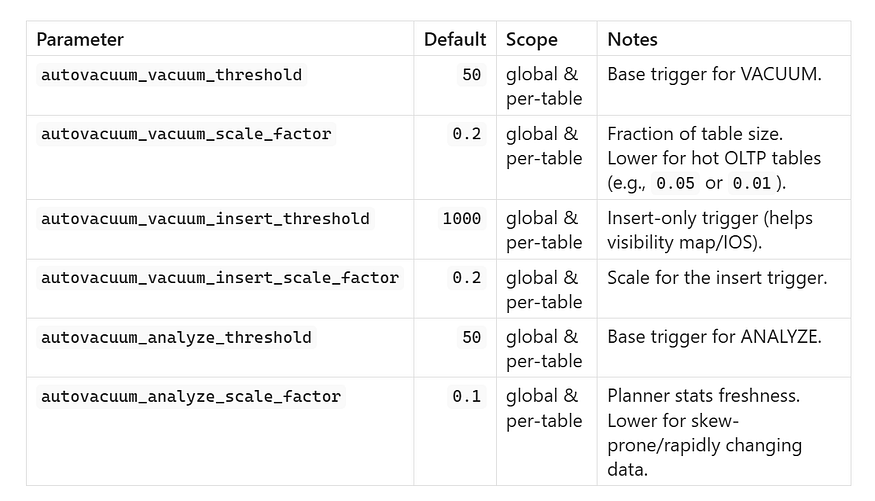
50 + (0.02 × 1,000,000)   
= 50 + 20,000   
= 20,050

👉 Autovacuum will run much earlier (after ~20k dead rows instead of 200k).

This keeps the table lean and improves query performance.

Tuning these values is critical for ****big, frequently updated tables**** to prevent bloat and keep performance stable.

Press enter or click to view image in full size



****Example (global)****

autovacuum\_vacuum\_threshold = 50  
autovacuum\_vacuum\_scale\_factor = 0.10  
autovacuum\_vacuum\_insert\_threshold = 1000  
autovacuum\_vacuum\_insert\_scale\_factor = 0.05  
autovacuum\_analyze\_threshold = 50  
autovacuum\_analyze\_scale\_factor = 0.05

****Example (per table)****

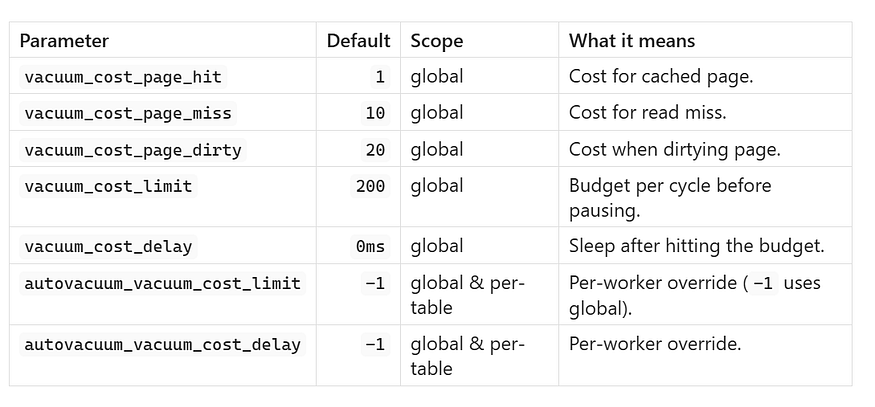
The command changes the ****autovacuum settings only for the orders table**** (it overrides the global defaults).

* ****autovacuum\_vacuum\_threshold = 100**** → Vacuum will trigger after at least 100 dead rows are found.
* ****autovacuum\_vacuum\_scale\_factor = 0.02**** → In addition, vacuum will trigger when dead rows exceed 2% of the table size.
* ****autovacuum\_analyze\_scale\_factor = 0.03**** → Analyze will refresh statistics when about 3% of the table changes, keeping query plans accurate.
* ****autovacuum\_vacuum\_insert\_threshold = 500**** → Vacuum will also trigger after 500 new rows are inserted (helps visibility for index-only scans).
* ****autovacuum\_vacuum\_insert\_scale\_factor = 0.03**** → Vacuum can also be triggered if inserts exceed 3% of the table size.

ALTER TABLE orders  
 SET (autovacuum\_vacuum\_threshold = 100,  
 autovacuum\_vacuum\_scale\_factor = 0.02,  
 autovacuum\_analyze\_scale\_factor = 0.03,  
 autovacuum\_vacuum\_insert\_threshold = 500,  
 autovacuum\_vacuum\_insert\_scale\_factor = 0.03);

## **3) Cost-based I/O throttling (pacing)**

Press enter or click to view image in full size



****Throughput-friendly pacing****

PostgreSQL’s VACUUM is ****I/O bound****. If it runs too aggressively, it can slow down user queries. To balance this, PostgreSQL uses a ****cost-based throttling system****:

* Each page access (hit, miss, dirty) has a “cost.”
* When the cost reaches vacuum\_cost\_limit, vacuum pauses for vacuum\_cost\_delay before continuing.
* This spreads out the work and avoids overwhelming disks.

Your configuration looks like this:

vacuum\_cost\_limit = 2000  
vacuum\_cost\_delay = 5ms  
autovacuum\_vacuum\_cost\_limit = 4000  
autovacuum\_vacuum\_cost\_delay = 2ms

## **🔹 **vacuum\_cost\_page\_hit = 1****

This is the cost charged when VACUUM touches a page already in shared buffers (cache hit).  
Default is ****1****. Raising it makes cached work “spend” the budget faster, leading to more frequent pauses.  
Best for smoothing I/O on very latency-sensitive OLTP systems where even cached scans should throttle.

👉 ****Example:**** With vacuum\_cost\_limit = 200, VACUUM can touch about ****200 ÷ 1 = 200**** cached pages before pausing.

## **🔹 **vacuum\_cost\_page\_miss = 10****

Cost when VACUUM must read a page from disk (cache miss).  
Default is ****10****. Increasing it penalizes reads more, forcing more frequent sleeps when the workload causes many misses.  
Best for HDD/slow storage or shared environments where reads spike latency.

👉 ****Example:**** With vacuum\_cost\_limit = 200, about ****200 ÷ 10 = 20**** disk reads happen before a pause.

## **🔹 **vacuum\_cost\_page\_dirty = 20****

Cost when VACUUM dirties a previously clean page (implies a write-back later).  
Default is ****20****. Raising it heavily discourages bursts of dirtying and spreads write pressure over time.  
Best when checkpoints or background writes are causing jitter and you want gentler write patterns.

👉 ****Example:**** With vacuum\_cost\_limit = 200, VACUUM can dirty ****200 ÷ 20 = 10**** pages before pausing. With limit ****2000****, that’s ****100**** pages.

## **🔹**vacuum\_cost\_limit = 2000

This means a ****manual VACUUM command**** can perform work that accumulates a cost of ****2000 units**** before it must pause.

* Default is much lower (200).
* Raising it allows VACUUM to clean more pages in one cycle, finishing faster.
* Best for when you can afford heavier maintenance windows (e.g., off-peak hours).

👉 Example: If scanning dirty pages (cost=20 each), VACUUM can process about 2000 ÷ 20 = 100 pages before pausing.

## **🔹**vacuum\_cost\_delay = 5ms

When the limit is hit (2000 units), VACUUM will pause for ****5 milliseconds**** before continuing.

* This short break reduces I/O pressure on the system.
* Default is 0ms (no pause).
* Setting a non-zero delay is safer for production, so user queries are less impacted.

## **🔹**autovacuum\_vacuum\_cost\_limit = 4000

This applies to ****autovacuum workers**** (the background vacuum processes).

* They get a ****higher cost limit**** (4000 vs. 2000), which means they can do ****twice as much work**** before pausing.
* This makes autovacuum run more efficiently, since background jobs are often expected to handle large workloads continuously.

## **🔹**autovacuum\_vacuum\_cost\_delay = 2ms

This sets the pause duration for autovacuum workers.

* At 2ms, it’s shorter than the 5ms used for manual VACUUM.
* That means autovacuum will pause less and move faster through large tables.
* This helps prevent bloat from growing too fast, especially on high-traffic tables.

## **✅ Putting It All Together**

With these settings:

* ****Manual VACUUM**** runs at a steady pace (2000 cost units, 5ms pause) → safer for on-demand use when you don’t want to overload the system.
* ****Autovacuum**** runs more aggressively (4000 cost units, 2ms pause) → ensures background cleanup keeps up with heavy workloads and prevents table bloat.

📌 ****Analogy:**** Imagine vacuuming your house.

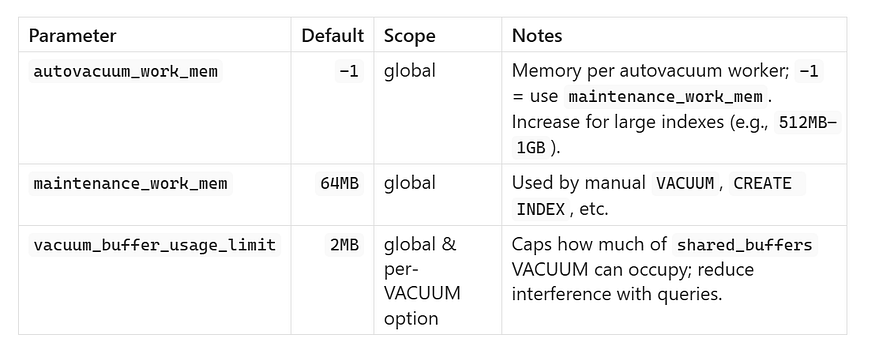
* Manual vacuuming (you doing it) → you take breaks more often (every 2000 units, 5ms rest).
* Autovacuum (robot vacuum) → it works harder and rests less (every 4000 units, 2ms rest), because you expect it to keep cleaning without you watching.

vacuum\_cost\_limit = 2000  
vacuum\_cost\_delay = 5ms  
autovacuum\_vacuum\_cost\_limit = 4000  
autovacuum\_vacuum\_cost\_delay = 2ms

Rule of thumb: If autovacuum hurts foreground latency, increase delays or reduce limits; if tables bloat, decrease delays / increase limits and/or increase workers.

## **4) Memory & buffer usage**

Press enter or click to view image in full size



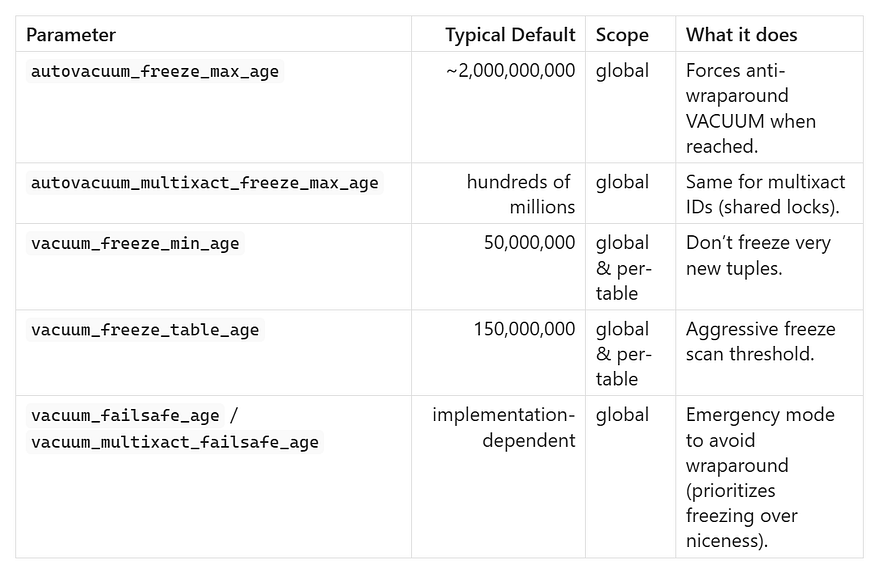
****Example****

autovacuum\_work\_mem = 1GB  
vacuum\_buffer\_usage\_limit = 8MB

## **5) Freezing & wraparound protection (critical)**

These protect you from transaction ID wraparound (catastrophic if ignored).

Press enter or click to view image in full size



****Aggressive one-off****

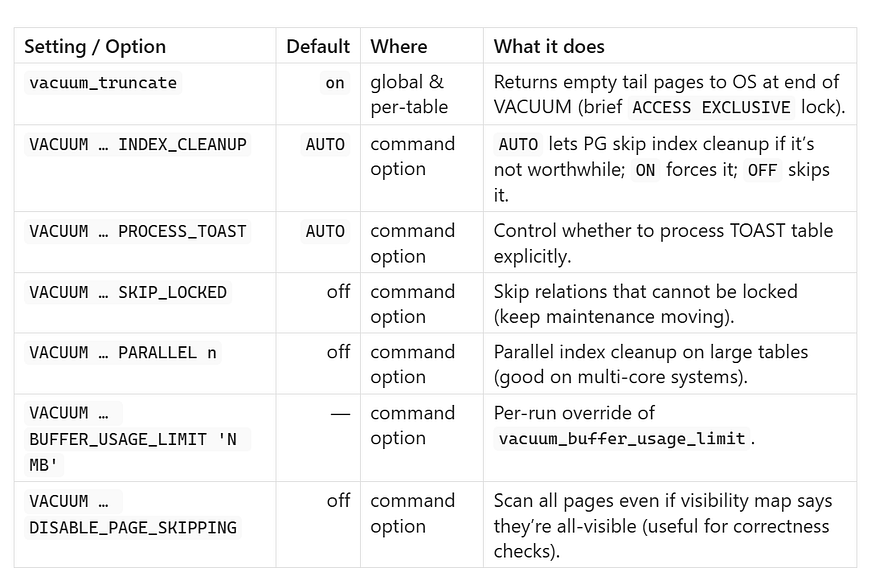
VACUUM (FREEZE, VERBOSE) my\_large\_table;

****Monitor risk****

SELECT  
 relname,  
 age(relfrozenxid) AS xid\_age  
FROM pg\_class c  
JOIN pg\_namespace n ON n.oid = c.relnamespace  
WHERE relkind = 'r'  
ORDER BY xid\_age DESC  
LIMIT 20;

## **6) Table truncation & index cleanup**

Press enter or click to view image in full size



## **Example 1:**

-- Force thorough index cleanup and shrink buffers used  
VACUUM (INDEX\_CLEANUP ON, BUFFER\_USAGE\_LIMIT '16MB') orders;

## **🔹 What it means:**

1. ****INDEX\_CLEANUP ON****

* Normally, PostgreSQL may skip cleaning indexes during vacuum if it thinks the benefit is small (default = AUTO).
* With ON, you force VACUUM to remove ****all dead index entries****.
* This helps avoid ****index bloat****, especially on large fact tables (orders) that get many updates or deletes.

2. ****BUFFER\_USAGE\_LIMIT '16MB'****

* This limits how much of shared\_buffers VACUUM is allowed to use during the operation.
* Default is very small (2MB).
* By setting it to 16MB, you let VACUUM use more memory to process pages, but not so much that it starves user queries.
* It keeps the maintenance predictable and avoids cache pollution.

👉 This command runs a vacuum on the big\_fact table, forcing a ****full index cleanup**** and allowing VACUUM to use ****up to 16MB of buffer cache**** for its work. It’s useful when you know indexes are bloated and want to reclaim space more aggressively.

## **Example 2:**

-- Keep maintenance moving under lock contention  
VACUUM (SKIP\_LOCKED, PARALLEL 4) orders;

## **🔹 What it means:**

1. ****SKIP\_LOCKED****

* If VACUUM can’t immediately lock a page (because another transaction is holding it), it will ****skip that page**** and continue with the rest.
* Without this, VACUUM might wait (or block), slowing down the process.
* This is useful in ****highly concurrent systems**** where you don’t want vacuum jobs to stall.

2. ****PARALLEL 4****

* Allows VACUUM to use ****4 parallel workers**** for index cleanup.
* Great for very large partitioned or heavily indexed tables like orders.
* It speeds up vacuuming on multi-core servers by splitting the work.

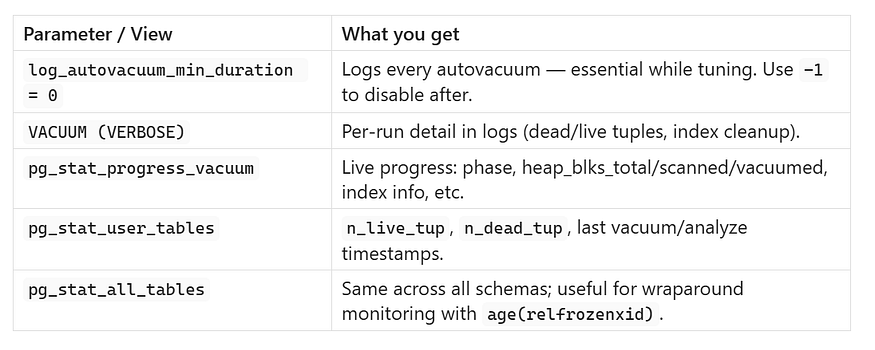
👉 This command vacuums the orders table but tells PostgreSQL ****not to wait on locked pages**** (SKIP\_LOCKED) and to use ****4 parallel workers**** for faster cleanup. It keeps the system responsive even under heavy load.

📌 ****Analogy:****

* The first is like deep-cleaning your house but only using a fixed number of cleaning supplies (16MB buffer) to stay organized.
* The second is like sending four cleaners into different rooms at once and telling them to ****skip locked rooms**** and come back later, so the cleaning doesn’t stall.

## **7) Logging & live monitoring**

Press enter or click to view image in full size



****Quick bloat watch****

SELECT relname, n\_dead\_tup, n\_live\_tup,  
 round(100.0\*n\_dead\_tup/GREATEST(n\_live\_tup,1),2) AS dead\_pct  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC  
LIMIT 20;

## **8) Per-table storage parameters (fine-grained control)**

Most autovacuum controls can be applied per table (and per TOAST table using the toast. prefix).

* autovacuum\_enabled (bool)
* autovacuum\_vacuum\_threshold (int)
* autovacuum\_vacuum\_scale\_factor (float)
* autovacuum\_vacuum\_insert\_threshold (int)
* autovacuum\_vacuum\_insert\_scale\_factor (float)
* autovacuum\_analyze\_threshold (int)
* autovacuum\_analyze\_scale\_factor (float)
* autovacuum\_vacuum\_cost\_limit (int)
* autovacuum\_vacuum\_cost\_delay (time)

****Examples****

-- Aggressive on a hot OLTP table  
ALTER TABLE payments  
 SET (autovacuum\_vacuum\_scale\_factor = 0.01,  
 autovacuum\_analyze\_scale\_factor = 0.02,  
 autovacuum\_vacuum\_cost\_limit = 8000,  
 autovacuum\_vacuum\_cost\_delay = 1ms);

-- Insert-only log table (favor visibility)  
ALTER TABLE http\_logs  
 SET (autovacuum\_vacuum\_insert\_threshold = 500,  
 autovacuum\_vacuum\_insert\_scale\_factor = 0.02,  
 autovacuum\_analyze\_scale\_factor = 0.02);

-- Temporarily disable (bulk backfill); wraparound vacuums still occur  
ALTER TABLE staging\_load SET (autovacuum\_enabled = off);

## **9) Practical presets (drop-in profiles)**

## **A) OLTP (many small updates/deletes)**

autovacuum = on  
autovacuum\_max\_workers = 6  
autovacuum\_naptime = 30s

autovacuum\_vacuum\_threshold = 50  
autovacuum\_vacuum\_scale\_factor = 0.02  
autovacuum\_analyze\_scale\_factor = 0.03  
autovacuum\_vacuum\_insert\_threshold = 500  
autovacuum\_vacuum\_insert\_scale\_factor = 0.03  
vacuum\_cost\_limit = 3000  
vacuum\_cost\_delay = 3ms  
autovacuum\_vacuum\_cost\_limit = 6000  
autovacuum\_vacuum\_cost\_delay = 1ms  
autovacuum\_work\_mem = 1GB  
vacuum\_buffer\_usage\_limit = 8MB  
log\_autovacuum\_min\_duration = 0

## **B) Insert-heavy (append-only logs, time-series)**

autovacuum\_vacuum\_insert\_threshold = 500  
autovacuum\_vacuum\_insert\_scale\_factor = 0.02  
autovacuum\_vacuum\_scale\_factor = 0.20 # normal dead-tuple trigger can stay higher  
autovacuum\_analyze\_scale\_factor = 0.05

## **C) Mixed workload, cautious pacing**

autovacuum\_max\_workers = 5  
autovacuum\_naptime = 45s  
vacuum\_cost\_limit = 2000  
vacuum\_cost\_delay = 5ms  
autovacuum\_vacuum\_scale\_factor = 0.05  
autovacuum\_analyze\_scale\_factor = 0.05  
log\_autovacuum\_min\_duration = 200ms

## **10) VACUUM command options (quick reference)**

VACUUM -- plain vacuum (no lock, space reusable)  
VACUUM (VERBOSE) -- detailed log  
VACUUM (FULL) -- rewrites table, exclusive lock, shrinks file  
VACUUM (FREEZE) -- aggressive freezing (xid safety)  
VACUUM (INDEX\_CLEANUP ON|OFF|AUTO)  
VACUUM (DISABLE\_PAGE\_SKIPPING) -- scan all pages regardless of VM  
VACUUM (PROCESS\_TOAST ON|OFF)  
VACUUM (SKIP\_LOCKED)  
VACUUM (PARALLEL 4)  
VACUUM (BUFFER\_USAGE\_LIMIT '16MB')

**VACUUM FULL**: use sparingly; it takes an **exclusive lock** and is **I/O heavy**. Prefer normal VACUUM + REINDEX for index bloat; consider CLUSTER/pg\_repack for online compaction if needed.

## **11) Tuning workflow (step-by-step)**

1. ****Measure****

* Identify hot/bloated tables: pg\_stat\_user\_tables (n\_dead\_tup, last\_autovacuum).
* Watch live runs: pg\_stat\_progress\_vacuum.
* Turn on logs: log\_autovacuum\_min\_duration = 0 (temporarily).

2. ****Adjust globally**** (small moves)

* Lower autovacuum\_vacuum\_scale\_factor (e.g., 0.2 → 0.1).
* Raise worker/memory/cost limits if hardware allows.

3. ****Target per-table outliers****

* Set aggressive per-table storage params for the top offenders.
* Add insert-trigger knobs for append-only tables.

4. ****Stabilize****

* Reduce log\_autovacuum\_min\_duration logging once happy.
* Keep an eye on age(relfrozenxid) for wraparound safety.

## **12) Your snippet + minimal must-adds**

You shared:

autovacuum = on  
autovacuum\_naptime = 60  
autovacuum\_vacuum\_scale\_factor = 0.2  
autovacuum\_analyze\_scale\_factor = 0.1

Add these ****minimum essentials****:

autovacuum\_vacuum\_threshold = 50  
autovacuum\_vacuum\_insert\_threshold = 1000  
autovacuum\_vacuum\_insert\_scale\_factor = 0.2  
log\_autovacuum\_min\_duration = 0 # enable during tuning, disable later